Constraint reasoning in Path-oriented Random Testing

Arnaud Gotlieb Matthieu Petit
INRIA Rennes - Bretagne Atlantique Research Centre
35042 Rennes Cedex, France
{*Arnaud.Gotlieb,Matthieu.Petit*}@*irisa.fr*

Abstract

Path-oriented Random Testing (PRT) aims at generating a uniformly spread out sequence of random test data that activate a single control flow path within an imperative program. The main challenge of PRT is to build efficiently such a test suite in order to minimize the number of rejects (test data that activate another control flow path). We address this problem with an original technique based on constraint reasoning over finite domains, a well-recognized Constraint Programming technique. Our approach derives path conditions by using symbolic execution and computes an approximation of their associated subdomain by using constraint propagation and constraint refutation.

1 Introduction

Random Testing is a fair testing technique that selects test data uniformly over the input domain of a program. Recently, we proposed to perform random testing at the path level, leading to this notion of Path-oriented Random Testing (PRT) [2]. PRT is an approach that derives the path conditions of a selected path by using symbolic execution and tries to build an uniform sequence of test data that activates only this path. The challenging problem consists in building *efficiently* a *uniform random test data generator* (URTG) by minimizing the number of rejects within the generated random sequence. A reject is produced whenever the randomly generated test datum does not satisfy the path conditions.

In this paper, we focus on the problem of minimizing the number of rejects in PRT by using constraint reasoning over finite domains. Our approach exploits subtle constraint propagation and constraint refutation over finite domains [3] to build an approximation of the input subdomain corresponding to the selected path. By reasoning on the constraints of path conditions, we remove parts of the input domain that are inconsistent with these constraints. The over-approximation should be as tight as possible in order to minimize the rejects during the test data generation. The shape of the over-approximation should also have the property of permitting easily to build an URTG. In addition, our approach is able to detect some non-feasible paths that cannot be identified with other Random Testing approaches such as adaptive RT [1] or feedback-directed RT [4].

Outline of the paper. Section 2 gives an overview of PRT based on constraint reasoning on a simple example. Section 3 gives some background on symbolic execution and random testing while Section 4 explains how we tuned usual Constraint Programming techniques to improve PRT. Section 5 details our algorithm to perform PRT. Finally, we conclude in Section 6.

2 A motivating example

Consider the C program of Fig.1 and the problem of building an URTG for path $1\to 2\to 3\to 4\to 5$. By looking at the decisions of the program, we see that x and y must range in $0..100$[1]. But, the other decisions cannot be tackled so easily. By using an URTG that independently picks up pairs (x_i, y_i) in $0..100 \times 0..100$ and rejects the pairs (x_j, y_j) that do not satisfy the constraints $y_j > x_j + 50 \wedge x_j * y_j < 60$, we get an URTG that solves the problem. However, this approach is highly expensive as it will reject a lot of randomly generated pairs. In fact, by manually analyzing the program, we see that the average probability of rejecting a pair is not far from $\frac{99}{100}$ with this approach. Indeed, activating the path $1\to 2\to 3\to 4\to 5$ has a very low probability as only 58 input points over 10201 satisfy the constraints. In contrast, by using constraint propagation and constraint refutation, we can minimize this probability and reduce the length of the generated test suite. By using constraint propagation over finite domains, we get immediately that any solution pair (x, y) must range over the rectangle

[1]Let suppose that **ush** stands for unsigned short integers

```
ush foo(ush x, ush y) {
  if (x < 101  &&  y < 101) {
      if (y > x + 50)
          ...
      if (x * y < 60)
          ...
```

Figure 1. Program foo

$D_1 = (x \in 0..1, y \in 51..100)$ which is a correct[2] and tight over-approximation of the solutions of the problem. Building a random test data generator for D_1 is easy as we can select x, y independently. This would not have been true if D_1 had the shape of a triangle, for example. In addition, by combining domain bisection and constraint refutation, we get an even tighter over-approximation. D_1 can be fairly divided into 4 subdomains: $(x = 0, y \in 51..75), (x = 1, y \in 51..75), (x = 0, y \in 76..100), (x = 1, y \in 76..100)$. This division is fair as each subdomain has exactly the same number of two-dimensional points. Thanks to constraint refutation, the fourth subdomain can be safely removed from the domain for which we want to build an URTG. Indeed, constraint propagation easily shows that there is no solution of the path conditions in this subdomain. As $D_2 = (x = 0, y \in 51..75) \cup (x = 0, y \in 76..100) \cup (x = 1, y \in 51..75)$ is the union of subdomains of same areas, we can still build an URTG for D_2 by selecting y independently from x. In fact, we design our method by keeping this latter constraint in mind. Finally, by using this method, the average probability of rejecting a possible pair in D_2 is just around $\frac{22}{100}$ (58 input points over the 75 of D_2 satisfy the two decisions).

3 Background

In this section, we recall how to derive the path conditions associated with a control flow path by using symbolic execution (Sec. 3.1) and the basic principles of Random Testing (Sec. 3.2).

3.1 Symbolic execution

Symbolic execution works by computing symbolic states for a selected path. A *symbolic state* for path $e \rightarrow n_1 \rightarrow \ldots \rightarrow n_k$ in program P is a triple $(e \rightarrow n_1 \rightarrow \ldots \rightarrow n_k, \{(v, \phi_v)\}_{v \in Var(P)}, PC)$ where ϕ_v is a symbolic expression associated with the variable v and $PC = c_1 \wedge \ldots \wedge c_n$ is a set of constraints associated with path $e \rightarrow n_1 \rightarrow \ldots \rightarrow n_k$, called the *path conditions*. $Var(P)$ denotes the set of variables in P. A symbolic expression is either a symbolic value (possibly **undef**) or a well parenthesized expression composed over symbolic values. In fact, when computing new symbolic expressions, each internal variable reference is replaced by its previously computed symbolic expression. Solving the path conditions yields either to show that the corresponding path is non-feasible or to find a test datum on which the path is executed.

[2] We did not loose any solution

3.2 Random Testing

Random Testing (RT) is the process of selecting test data at random according to an uniform probability distribution over the program's input domain. The input domain of the program under test is formed by the Cartesian product of bounded intervals of integers. Technically, such an input domain is called an hypercuboid, which is the n-dimensional extension of the 3-dimensional cuboid. Performing random testing based on an uniform distribution over an hypercuboid domain is simple as any of its points can be randomly chosen by selecting its coordinates independently.

Our PRT approach makes use of two fundamental invariance properties of uniform generators. The first property states that an uniform random generator for a given domain D can also serve as an uniform generator for any of the subdomains of D. More formally:

Property 1 (First invariance property) *Let S be a sequence of uniformly distributed tuples of values for a domain D, then for any subset D' of D, it is always possible to extract from S a sequence S' of uniformly distributed tuples for D'.*

Extracting such a sequence from S can be done by rejecting the tuples that do not belong to D'. The remaining sequence S' is still uniformly spread out over D' as D' is a subset of D. To obtain an uniform sequence of values for D', it suffices to examine each of the tuples of the sequence for D and to reject those tuples that do not belong to D'. Of course, the smaller D' w.r.t. D, the larger the uniform sequence for D must be. The second property states that an URTG can be built in a hierarchical manner:

Property 2 (Second invariance property) *Let D be a domain of n tuples, let k be a divisor of n and $D_1, \ldots, D_k$ be a partition of D such that each D_i possesses the same number of tuples, then an uniform random sequence for D can be built by generating first a uniform random sequence over $D_1, \ldots, D_k$, and then picking up a single tuple in each D_i, at random.*

The important point here is that all the domains D_i have the same number of tuples.

4 Constraint Reasoning in PRT

Path-oriented Random Testing aims at finding a test suite that uniformly exercises a selected control flow path. We propose using constraint reasoning to build efficiently such a test suite. Constraint reasoning usually involves two interleaved processes in order to get a solution of a constraint system: constraint propagation and variable labelling. Constraint propagation prunes the variation domain of variables by eliminating inconsistent values while labelling tries to infer solutions by elaborating hypothesis and refuting subdomains. In our approach we exploit constraint propagation (Sec.4.1) and constraint refutation (Sec.4.2).

4.1 Constraint propagation

The process. Constraint propagation introduces constraints from the path conditions into a propagation queue. Then, an iterative algorithm manages each constraint one by one into this queue by filtering the domains of variables of their inconsistent values. When the variation domain of variables is large, filtering algorithms consider usually only the bounds of the domains for efficiency reasons: a domain $D = \{v_1, v_2, \ldots, v_{n-1}, v_n\}$ is approximated by the range $v_1..v_n$. When the domain of a variable is pruned then the algorithm reintroduces in the queue all the constraints where this variable appears to propagate this information. The algorithm iterates until the queue becomes empty, which corresponds to a state where no more pruning can be performed. When selected in the propagation queue, each constraint is added into a constraint–store which memorizes all the considered constraints. The constraint–store is contradictory if the domain of at least one variable becomes empty. In this case the corresponding path is shown to be non-feasible.

Efficiency. When considering only the bounds of domains, constraint propagation is really very efficient as it runs in $O(m)$ where m denotes the number of constraints [3]. But, it is worth noticing that constraint propagation alone does not guarantee satisfiability. Consider the following constraint system over finite domains: $x \in 1..100, y \in 1..100, z \in 1..100, x = y * z, x < z * y$. On this example constraint propagation does not perform any pruning on the domains, although the constraint system is clearly unsatisfiable. Hopefully, these situations are infrequent in practice and inconsistent subdomains can often be discarded.

Hypercuboids. Constraint propagation over finite domain variables computes *hypercuboids*: each variable of an n-dimensional space belongs to a range $Min..Max$ of values. In the example of Fig.1, constraint propagation permits to get the hypercuboid $D_1 = (x \in 0..1, y \in 51..100)$ where D_1 is an over-approximation of the solution set of the path conditions $x \in 0..100, y \in 0..100, y > x + 50 \wedge x * y < 60$.

4.2 Constraint refutation

Constraint refutation is the process of temporarily adding a constraint to a set of constraints and testing whether the resulting constraint system has no solution by using constraint propagation. If the resulting constraint system is unsatisfiable, the added constraint is shown to be contradictory with the rest of the constraints and then it is refuted. When constraint propagation does not yield to a contradiction, then nothing can be deduced as constraint propagation is incomplete in general. Based on constraint addition/removal and propagation, this process is very efficient and it can be exploited in PRT to test domain intersection: let D be a subdomain defined by a set of constraints and C be a constraint, checking whether $D \cap C = \emptyset$ is true can be done by adding constraint C to D and test whether C is refuted or not. An example of such a refutation was given in the motivating example of the paper.

5 PRT based on constraint reasoning

In this section, we detail our algorithm to perform PRT based on constraint reasoning. Firstly, we detail how to fairly divide the hypercuboid resulting from constraint propagation (Sec. 5.1) and secondly, we explain how to exploit constraint refutation to prune the subdomain associated with the path conditions (Sec.5.2). Finally, we give our algorithm that exploits both these processes to build an efficient URTG for PRT (Sec.5.3).

5.1 Dividing the hypercuboid

Applying constraint propagation on the path conditions results in an hypercuboid that is a correct approximation of the solution set of the path conditions. Using this approximation to define an URTG for PRT is possible but not optimal. We propose a new way of refining this hypercuboid in smaller subdomains. It is worth noticing that special attention must be paid to the way this hypercuboid is broken into subdomains in order to preserve the uniformity of the generator. Let k be a given parameter, called the *division parameter*, our method is based on the division of each variable domain into k subdomains of equal area. When the size of a domain variable cannot be divided by k, then we enlarge its domain until its size can be divided by k. By iterating this process over all the n input variables, we get a fair partition of the (augmented) hypercuboid, in k^n subdomains.

5.2 Pruning the hypercuboid

As said previously, constraint refutation can be used to test efficiently domain intersection. Thus, we eliminate parts of the hypercuboid that are inconsistent with the path conditions. From a domain D, we can safely eliminate subdomains that are not consistent with the rest of the constraints, leading to a smaller domain D'.

And by using the second invariance property, we get an easy way to draw uniformly test data. It suffices to draw at random a subdomain in D' and then to draw at random a value in this subdomain.

Using constraint refutation has another advantage as it help detecting non-feasible paths. Recall that non-feasible paths correspond to unsatisfiable constraint systems. Hence, when all the subdomains of the partition are shown to be inconsistent, then it means that the corresponding path is non-feasible. This contrasts with RT approaches such as Adaptive RT [1] or Feedback-directed RT [4] which cannot detect non-feasible paths. Note however that our approach can miss to detect some non-feasible paths due to the incompleteness of constraint propagation.

5.3 The algorithm based on constraint reasoning

We present an algorithm that performs PRT based on constraint reasoning. The algorithm takes as inputs a set of variables along with their variation domain, PC a constraint set corresponding to the path conditions of the selected path, k the division parameter, and N the length of the expected random sequence. The algorithm returns a list of N uniformly distributed random tuples that all satisfy the path conditions. The list is void when the corresponding path is detected as being non-feasible.

Firstly, the algorithm partitions the hypercuboid resulting from constraint propagation in k^n subdomains of equal area (`Divide` function). Then, each subdomain D_i in the partition is checked for unsatisfiability. This results in a list of subdomains $D'_1, \ldots, D'_p$ where $p \leq k^n$. Secondly, an URTG is built from this list by picking up first a subdomain and then picking up a tuple inside this subdomain. If the selected tuple does not satisfy the path conditions then it is simply rejected. This process is repeated until a sequence of N test data is generated. This algorithm is semi-correct, meaning that when it terminates, it is guaranteed to provide the correct expected result, but it is not guaranteed to terminate. Indeed, in the second loop, N is decreased iff t satisfies PC, which can happen only if PC is satisfiable. In other words, if PC is unsatisfiable and if this has not been detected by constraint propagation ($p \geq 1$), then the algorithm will not terminate. Note that similar problems arise with random testing or path testing as nothing prevents an unsatisfiable goal PC to be selected and, in this case, all the test cases will be rejected. In practice, a time out mechanism is necessary to enforce termination.

Algorithm 1: Path-oriented Random Testing

```
Input  : (x_1,..,x_n), PC, k, N
Output : t_1,..,t_N or ∅ (non-feasible path)

T := ∅ ;
(D_1,..,D_{k^n}) := Divide({x_1,..,x_n}, k);
forall D_i ∈ (D_1,...,D_{k^n}) do
    if D_i is inconsistent w.r.t. PC then
        remove D_i from (D_1,..,D_{k^n}) ;
    end
end
Let D'_1,..,D'_p be the remaining list of domains;
if p ≥ 1 then
    while N > 0 do
        Pick up uniformly D at random from
        D'_1,..,D'_p;
        Pick up uniformly t at random from D;
        if PC is satisfied by t then
            add t to T ;
            N := N − 1;
        end
    end
end
return T;
```

6 Conclusion

This paper presents an approach (Path-oriented Random Testing) that combines both the advantages of path testing and random testing, through the usage of constraint reasoning over finite domains. We have proposed to exploit subtle constraint reasoning over finite domains to build efficiently a uniform sequence of test data that exercize a selected path of a program. Perspectives of this work include the experimental evaluation of this technique of industrial-sized C programs.

References

[1] T. Y. Chen, H. Leung, and I. K. Mak. Adaptive random testing. In *ASIAN'04*, pages 320–329, 2004.

[2] A. Gotlieb and M. Petit. Path-oriented random testing. In *1st ACM Int. Workshop on Random Testing (RT'06)*, Portland, Maine, July 2006.

[3] P. Hentenryck, V. Saraswat, and Y. Deville. Design, implementation, and evaluation of the constraint language cc(fd). *Journal of Logic Programming*, 37:139–164, 1998.

[4] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball. Feedback-directed random test generation. In *Proc. of the Int. Conf. on Software Engineering (ICSE'07)*, Minneapolis, 2007.